Safe Compositional Specification of Networking Systems*
TRAFFIC The Language and Its Type Checking

Likai Liu, Assaf J. Kfoury, Azer Bestavros, Adam D. Bradley, Yarom Gabay, and Ibrahim Matta
{liulk,kfoury,best,artdodge,yarom,matta}@cs.bu.edu
Department of Computer Science, Boston University

May 12, 2005

Abstract

This paper formally defines the operational semantic for TRAFFIC, a specification language for flow composition applications proposed in [3], and presents a type system based on desired safety assurance. We provide proofs on reduction (weak-confluence, strong-normalization and unique normal form), on soundness and completeness of type system with respect to reduction, and on equivalence classes of flow specifications. Finally, we provide a pseudo-code listing of a syntax-directed type checking algorithm implementing rules of the type system capable of inferring the type of a closed flow specification.

1 Introduction

In a previous paper [2], we stated that the combined heterogeneity and dynamic open nature of network systems makes composition quite challenging, and thus programming network services has been largely inaccessible to the average user. With this in mind, we identified and outlined a research agenda in which we aim to develop a specification language that is expressive enough to describe different components of a network service, and that will include type hierarchies inspired by type systems in general programming languages that enable the safe composition of software components.

Safe composition of network flows is a collaborative effort combining ideas and methods from two research areas: networking and programming languages. This work was initially motivated by the need to devise a formalism to describe Quality of Service (QoS) properties of flows and rigorously establish conditions under which flows can be composed safely. The formalism to verify the safety of networked controller compositions comes in the form of a language, a type system, and a type inference algorithm. The language provides a way to describe flow compositions. The type system provides rules of the safety properties. Finally, the type inference algorithm provides an automatic deduction of types resulting from composing flows. These three parts together enable us to build a tool that mechanically analyzes and verifies safety properties of network flows.

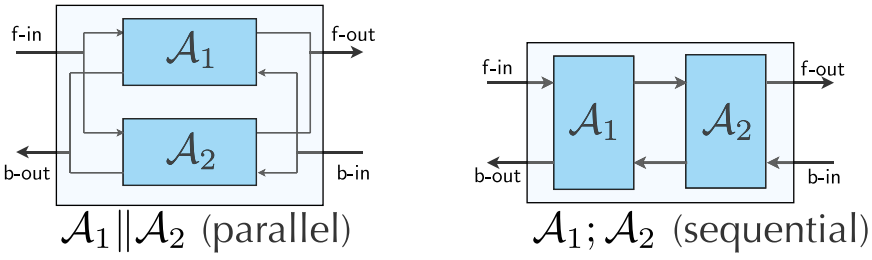
This paper is written with an emphasis from the programming languages perspective—we develop a system that can mechanically infer properties and results from types like those sketched in [3]; we use the specification language tailored for flow composition applications and rigorously specify rules for the construction and inference of types based upon the elements of those applications. We begin by describing a conceptual structure for *flows*—the building blocks from which composite applications are constructed—and rules governing the composition of such flows. We then present a syntax for which types are assigned to flows and formalize safety properties of the specification language using types. Finally, we provide a type inference algorithm that performs syntax-driven analysis to deduce the resulting type of a flow.

2 The Language of Flows

A flow is a unit of building blocks that constitute a networked system for a given application. A flow building block is characterized by four connection sockets representing input and output channels of forward (signaling) and backward

*This work was supported in part by NSF grants ITR ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067.

Figure 2.1: Parallel and Sequential Composition of Flows



(a) Parallel Composition

(b) Sequential Composition

(feedback) directions. Composition of flows are either parallel (i.e., aggregating input and output sockets of two flows, as in Figure 2.1(a)) or sequential (i.e., connecting the output socket of one flow to the input socket of the next flow, as in Figure 2.1(b)).

Both the forward and backward channels are typed sockets according to desired properties for a given application. The forward channel may include such properties as its convergence (or lack thereof), rise time, settle time, whether or not it is over-damped, its steady-state error, etc. The backward channel is similarly typed to reflect the origins of the feedback signals it carries, the range of latency jitter the feedback signal(s) experience, etc. As such, a flow is analogous to a function which takes two arguments (the incoming sockets) and returns two values (the outgoing sockets), and we would expect the typing of flows to reflect such a functional relationship. Any "type" we assign to or infer for a flow corresponds to a 4-tuple of types describing the allowable inputs and range of outputs that controller is compatible with.

2.1 Specification of Global Flows

When we speak of an application, we refer to applications that make use of end-to-end flows built upon network nodes that are also called flow controllers. For simplicity, and with moderate trade-off in expressiveness, we provide two ways to compose end-to-end flows, namely, sequential composition and parallel composition. Both compositions are binary operators, so they combine any two flows. Network nodes that are not a part of the flow are not being considered as a part of an application. Since flows are end-to-end, the language needs not address issues of network topology, therefore sequential and parallel compositions are mostly sufficient.

Flow composition consists of three kinds of flow: local flows, flow variables, and global-flows. A local flow is a singleton component of a flow controller, for which complete type information is provided. A flow variable is a placeholder representing flows which are yet to be fully specified. Global-flow is a flow composed either sequentially or in parallel from local flows, flow variables, or other global-flows. Furthermore, the language has a syntax for "let-binding," which specifies what is to be filled in place of a flow-variable. Let-binding and flow variables together give us the facility to abstract flow expressions.

We specify global flows syntactically using the following BNF:

$$
\begin{array}{rcll}
x, y, z & \in & \mathsf{FlowVar} & \text{flow variable} \\
A, B, C & \in & \mathsf{LocalFlow} & \text{local flow} \\
\mathcal{A}, \mathcal{B}, \mathcal{C} & \in & \mathsf{GlobalFlow} ::= A \mid x & \\
 & & \mid \mathcal{A}; \mathcal{B} & \text{sequential flow} \\
 & & \mid \mathcal{A} \| \mathcal{B} & \text{parallel flow} \\
 & & \mid \mathbf{let}\ x = \mathcal{A}\ \mathbf{in}\ \mathcal{B} & \text{let-binding}
\end{array}
$$

A *parallel flow*, denoted $(\mathcal{A} \| \mathcal{B})$ and illustrated in Figure 2.1(a) in which two flows are placed (logically) parallel, offering simultaneous rather than serial service and data flow. A *sequential flow*, denoted $(\mathcal{A}; \mathcal{B})$, is a composition like that in Figure 2.1(b) in which two flows are placed (logically) adjacently and joined. The sequential operator ";" and the parallel operator "$\|$" have the same precedence, and both associate to the left.

The *operational semantic* of the language, i.e., the notion of reduction, is based on substitution, whose purpose is

to substitute a global-flow expression for flow variable occurrences in the case of a let-binding. Note that let-binding is closely related to β-redexes in the λ-calculus. Although TRAFFIC resembles a general-purpose language where the program executes by reduction rules, TRAFFIC is a specification language that differs from a general purpose language in the sense that reduction happens during the design stage: it corresponds to the act of assembling flow controllers. A specification written in TRAFFIC does not execute in the traditional sense.

Not every flow variables necessarily have a corresponding let-binding, which we formalize by the notion of *free variables* defined below.

Definition 2.1 (Free Variable Occurrences). Given a global-flow specification $\mathcal{A}$, the free variable occurrences in $\mathcal{A}$, written as $FV(\mathcal{A})$, is defined by

$$FV(\mathcal{A}) \quad = \quad \begin{cases} \varnothing & \text{if } \mathcal{A} = A \\ \{x\} & \text{if } \mathcal{A} = x \\ FV(\mathcal{A}_1) \cup FV(\mathcal{A}_2) & \text{if } \mathcal{A} = \mathcal{A}_1; \mathcal{A}_2 \text{ or } \mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2 \\ FV(\mathcal{B}_0) \cup (FV(\mathcal{B}) - \{x\}) & \text{if } \mathcal{A} = \mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B} \end{cases}$$

We say that a flow $\mathcal{A}$ is *closed* iff $FV(\mathcal{A}) = \varnothing$ (either it does not contain flow variables, or that every flow variables in the body of a let-binding are bound to a corresponding **let** statement); otherwise, we say a flow is *open*. If a flow is closed, we can reduce it to another flow consisting of only sequential and parallel compositions of local flows. Intuitively speaking, a closed flow consisting of let-bindings and flow variables can be flattened to simple compositions of local flow controllers. This is also the case where analysis and verification of types can be easily performed on the flow.

2.2 Reduction

In order to reason about flow variables as gaps in a global-flow specification, we must formalize what it means for those gaps to be filled. This is done by defining *substitution* on flow variables: Substituting $\mathcal{A}$ for x in $\mathcal{B}$ is written $[x := \mathcal{A}]\mathcal{B}$. The notion is made precise by induction on the definition of flows:

$$\begin{aligned} [x := \mathcal{A}]y &= \begin{cases} \mathcal{A} & \text{if } x = y, \\ y & \text{if } x \neq y, \end{cases} \\ [x := \mathcal{A}]A &= A, \\ [x := \mathcal{A}](\mathcal{B}; \mathcal{C}) &= ([x := \mathcal{A}]\mathcal{B}); ([x := \mathcal{A}]\mathcal{C}), \\ [x := \mathcal{A}](\mathcal{B} \| \mathcal{C}) &= ([x := \mathcal{A}]\mathcal{B}) \| ([x := \mathcal{A}]\mathcal{C}), \\ [x := \mathcal{A}](\mathbf{let}\ \ y = \mathcal{B}\ \ \mathbf{in}\ \ \mathcal{C}) &= \mathbf{let}\ \ y' = [x := \mathcal{A}]\mathcal{B}\ \ \mathbf{in}\ \ [x := \mathcal{A}][y := y']\mathcal{C} \quad y' \text{ is fresh} \end{aligned}$$

Note that the rules of substitution have two important properties: (1) in $[x := \mathcal{A}](\mathbf{let}\ \ y = \mathcal{B}\ \ \mathbf{in}\ \ \mathcal{C})$, $\mathcal{A}$ is not substituted for a bound occurrence of x in the subexpression $\mathcal{C}$ if $x = y$, and (2) no free occurrence of y in $\mathcal{A}$, if any, is captured by "$\mathbf{let}\ \ y = \mathcal{B}\ \ \mathbf{in}\ \ \ldots$" after the substitution. This ensures that unbound variables remain unbound and that variables remain bounded to the appropriate let-binding after substitution.

Our notion of *reduction*, denoted $\rightarrow$, on global-flow specifications is a binary relation defined by the following axiom of a redex:

$$(\mathbf{let}\ \ x = \mathcal{A}\ \ \mathbf{in}\ \ \mathcal{B}) \rightarrow [x := \mathcal{A}]\mathcal{B}$$

and inference rules:

$$\frac{\mathcal{A} \rightarrow \mathcal{A}'}{\mathcal{A}; \mathcal{B} \rightarrow \mathcal{A}'; \mathcal{B}} \qquad \frac{\mathcal{B} \rightarrow \mathcal{B}'}{\mathcal{A}; \mathcal{B} \rightarrow \mathcal{A}; \mathcal{B}'}$$

$$\frac{\mathcal{A} \rightarrow \mathcal{A}'}{\mathcal{A} \| \mathcal{B} \rightarrow \mathcal{A}' \| \mathcal{B}} \qquad \frac{\mathcal{B} \rightarrow \mathcal{B}'}{\mathcal{A} \| \mathcal{B} \rightarrow \mathcal{A} \| \mathcal{B}'}$$

$$\frac{\mathcal{A} \rightarrow \mathcal{A}'}{(\mathbf{let}\ \ x = \mathcal{A}\ \ \mathbf{in}\ \ \mathcal{B}) \rightarrow (\mathbf{let}\ \ x = \mathcal{A}'\ \ \mathbf{in}\ \ \mathcal{B})}$$

$$\frac{\mathcal{B} \rightarrow \mathcal{B}'}{(\mathbf{let}\ \ x = \mathcal{A}\ \ \mathbf{in}\ \ \mathcal{B}) \rightarrow (\mathbf{let}\ \ x = \mathcal{A}\ \ \mathbf{in}\ \ \mathcal{B}')}$$

Finally, let $\xrightarrow{*}$ denote the reflexive, transitive closure of $\rightarrow$, as a shorthand for zero or more steps of reduction.

Note that reduction may occur in any part of the global-flow specification as long as there is a redex. In the case where multiple redexes are present, the order of reduction may be arbitrary, so reduction sequences are not deterministic. Despite the non-determinism, we show that reduction terminates, and that every global-flows are reduced to a unique result.

We explain formally what it means for reduction to terminate: a global-flow specification $\mathcal{A}$ is said to be in *normal form* if $\mathcal{A}$ contains no redex (in our case, no let-binding), so $\mathcal{A}$ cannot be reduced further. We say that $\mathcal{A}$ has a normal form if there exists $\mathcal{B}$ such that $\mathcal{A} \xrightarrow{*} \mathcal{B}$ and $\mathcal{B}$ is in normal form, in which case we also say that $\mathcal{A}$ *normalizes* to $\mathcal{B}$. We say that $\rightarrow$ *strongly normalizes* if all reduction sequences are finite for every global-flow specification $\mathcal{A}$.

We will show that reduction of our global-flow specification language is *strongly normalizing*. The normal form of $\mathcal{A}$ is used to reason about the soundness (Theorem 4.2) and completeness (Theorem 4.4) of typing rules in Section 4. Furthermore, even though the reduction sequence for $\mathcal{A}$ is not deterministic, the normal form of $\mathcal{A}$ is unique. Specifications can be classified by letting the equivalence class be a set of specifications that normalize to the same unique normal form. If $\mathcal{A}$ is typable, then all global-flow specifications in the same equivalence class as $\mathcal{A}$ are also typable and have the same type. This will also be shown (by type soundness and completeness) once we formalize types of flows.

We first show that reduction satisfies *weak confluence* and *strong normalization* (reduction always halts). We use both to show *confluence* (reduction converges) and again, by *strong normalization*, conclude that normal form is unique.

Weak Confluence We proceed to show that reducing global-flow specifications satisfies weak confluence, formalized below.

Fact 2.2 (Reduction around Substitution). *Let $\mathcal{A}$, $\mathcal{A}'$ be global-flow specifications such that $\mathcal{A} \rightarrow \mathcal{A}'$, then for any $\mathcal{B}$, we can have:*

1. $[x := \mathcal{A}]\mathcal{B} \xrightarrow{*} [x := \mathcal{A}']\mathcal{B}$ *(reduction on the substitute).*
2. $[x := \mathcal{B}]\mathcal{A} \rightarrow [x := \mathcal{B}]\mathcal{A}'$ *(reduction on the substrate).*

Proof. The first is done by applying, on each instance of $\mathcal{A}$ in $[x := \mathcal{A}]\mathcal{B}$, the reduction steps for $\mathcal{A} \rightarrow \mathcal{A}'$. The number of steps from $[x := \mathcal{A}]\mathcal{B}$ to $[x := \mathcal{A}']\mathcal{B}$ is the number of free occurrences of x in $\mathcal{B}$. The second is done by keeping the instances of $\mathcal{B}$ intact while reducing $[x := \mathcal{B}]\mathcal{A}$ around $\mathcal{B}$ in the same manner as $\mathcal{A} \rightarrow \mathcal{A}'$. The number of steps from $[x := \mathcal{B}]\mathcal{A}$ to $[x := \mathcal{B}]\mathcal{A}'$ is exactly one. □

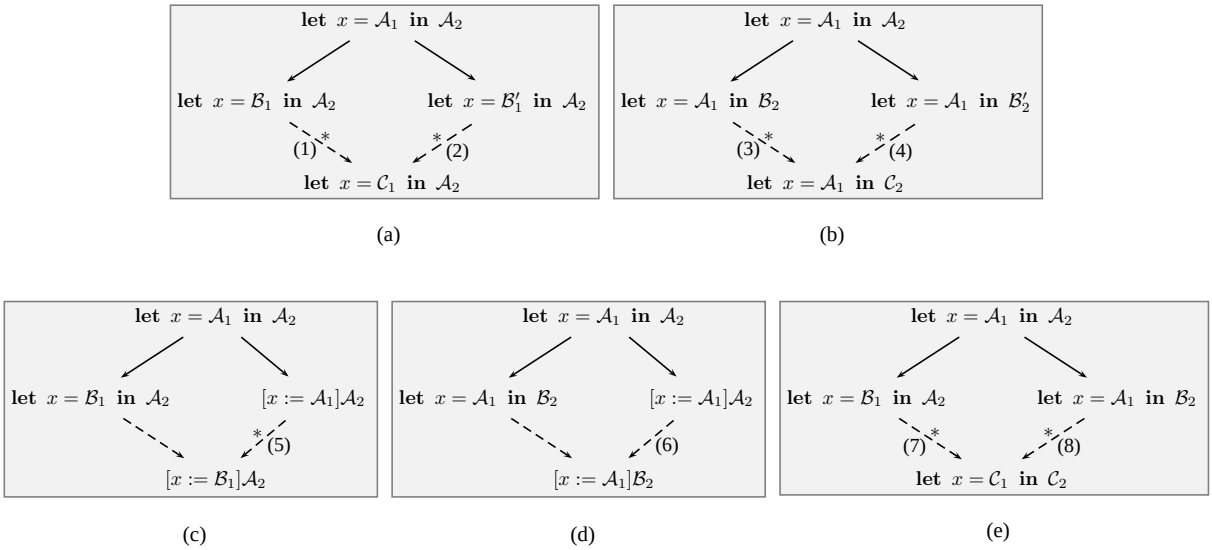
Lemma 2.3 (Weak Confluence). *Let $\mathcal{A}$, $\mathcal{B}$, $\mathcal{B}'$ and $\mathcal{C}$ be global-flow specifications such that $\mathcal{A} \rightarrow \mathcal{B}$ and $\mathcal{A} \rightarrow \mathcal{B}'$. Then there is a global-flow specification C such that $\mathcal{B} \xrightarrow{*} \mathcal{C}$ and $\mathcal{B}' \xrightarrow{*} \mathcal{C}$.*

Proof. By analyzing the possible forms of a global-flow specification.

- case $\mathcal{A}$ is A, a local-flow is already in normal form.
- case $\mathcal{A}$ is x, is also already in normal form.
- case $\mathcal{A}$ is $\mathcal{A}_1; \mathcal{A}_2$. Using induction hypothesis on $\mathcal{A}_1$, suppose there is $\mathcal{B}_1$ and $\mathcal{B}'_1$ such that $\mathcal{A}_1 \rightarrow \mathcal{B}_1$ and $\mathcal{A}_1 \rightarrow \mathcal{B}'_1$, then there is a $\mathcal{C}_1$ such that $\mathcal{B}_1 \xrightarrow{*} \mathcal{C}_1$ and $\mathcal{B}'_1 \xrightarrow{*} \mathcal{C}_1$. Again, using induction hypothesis on $\mathcal{A}_2$, suppose there is $\mathcal{B}_2$ and $\mathcal{B}'_2$ such that $\mathcal{A}_2 \rightarrow \mathcal{B}_2$ and $\mathcal{A}_2 \rightarrow \mathcal{B}'_2$, then there is a $\mathcal{C}_2$ such that $\mathcal{B}_2 \xrightarrow{*} \mathcal{C}_2$ and $\mathcal{B}'_2 \xrightarrow{*} \mathcal{C}_2$. Let $\mathcal{B}$ and $\mathcal{B}'$ be any of $\mathcal{B}_1; \mathcal{B}_2$, $\mathcal{B}'_1; \mathcal{B}_2$, $\mathcal{B}_1; \mathcal{B}'_2$, $\mathcal{B}'_1; \mathcal{B}'_2$, and let $\mathcal{C}$ be $\mathcal{C}_1; \mathcal{C}_2$. It is clear that both $\mathcal{B} \xrightarrow{*} \mathcal{C}$ and $\mathcal{B}' \xrightarrow{*} \mathcal{C}$.
- case $\mathcal{A}$ is $\mathcal{A}_1 \| \mathcal{A}_2$. Same as above, there is a $\mathcal{C} = \mathcal{C}_1 \| \mathcal{C}_2$ that all intermediate reduction steps of $\mathcal{A}_1$ and $\mathcal{A}_2$ converge to.
- case $\mathcal{A}$ is $\mathbf{let}\ x = \mathcal{A}_1\ \mathbf{in}\ \mathcal{A}_2$. This is best illustrated with commutative diagrams in Figure 2.2, each representing possible choices of $\mathcal{B}$ and $\mathcal{B}'$, with the outcome $\mathcal{C}$. The diagram edges that are not straightforward reductions are labeled and explained below.

  Let claim (i) be induction hypothesis on $\mathcal{A}_1$, that is, suppose there is $\mathcal{B}_1$ and $\mathcal{B}'_1$ such that $\mathcal{A}_1 \rightarrow \mathcal{B}_1$ and $\mathcal{A}_1 \rightarrow \mathcal{B}'_1$, then there is a $\mathcal{C}_1$ such that $\mathcal{B}_1 \xrightarrow{*} \mathcal{C}_1$ and $\mathcal{B}'_1 \xrightarrow{*} \mathcal{C}_1$. Let claim (ii) be induction hypothesis on $\mathcal{A}_2$,

Figure 2.2: Weak confluence diagrams for the **let** case.



so suppose there is $\mathcal{B}_2$ and $\mathcal{B}'_2$ such that $\mathcal{A}_2 \to \mathcal{B}_2$ and $\mathcal{A}_2 \to \mathcal{B}'_2$, then there is a $\mathcal{C}_2$ such that $\mathcal{B}_2 \xrightarrow{*} \mathcal{C}_2$ and $\mathcal{B}'_2 \xrightarrow{*} \mathcal{C}_2$.

Edges (1) and (2) are consequences of claim (i); (3) and (4) are consequences of claim (ii). Edge (5) is a consequence of Fact 2.2(1); edge (6) a consequence of Fact 2.2(2).

Edge (7) represents the reduction sequence,

$$\begin{aligned} \textbf{let } x = \mathcal{B}_1 \textbf{ in } \mathcal{A}_2 &\xrightarrow{*} \textbf{let } x = \mathcal{C}_1 \textbf{ in } \mathcal{A}_2 \quad \text{by claim (i)} \\ &\xrightarrow{*} \textbf{let } x = \mathcal{C}_1 \textbf{ in } \mathcal{C}_2 \quad \text{by claim (ii)} \end{aligned}$$

and edge (8) the sequence,

$$\begin{aligned} \textbf{let } x = \mathcal{A}_1 \textbf{ in } \mathcal{B}_2 &\xrightarrow{*} \textbf{let } x = \mathcal{A}_1 \textbf{ in } \mathcal{C}_2 \quad \text{by claim (ii)} \\ &\xrightarrow{*} \textbf{let } x = \mathcal{C}_1 \textbf{ in } \mathcal{C}_2 \quad \text{by claim (i).} \end{aligned}$$

The choices of $\mathcal{B}_1$ or $\mathcal{B}'_1$ and $\mathcal{B}_2$ or $\mathcal{B}'_2$ in the intermediate step does not matter.

□

Strong Normalization Here we present a simple proof inspired by [1, pp. 288]. For every flow variable $x \in \mathsf{FlowVar}$, we annotate a weight $n \in \mathbb{N}$ to the superscript of x, written as x^n. A flow is *weighted* by assigning a weight to every flow variables in it. The weight of a weighted flow is defined by:

$$w(\mathcal{A}) \quad = \quad \begin{cases} 0 & \text{if } \mathcal{A} = A \\ n & \text{if } \mathcal{A} = x^n \\ w(\mathcal{A}_1) + w(\mathcal{A}_2) & \text{if } \mathcal{A} = \mathcal{A}_1; \mathcal{A}_2 \text{ or } \mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2 \text{ or} \\ & \quad \mathcal{A} = \textbf{let } x = \mathcal{A}_1 \textbf{ in } \mathcal{A}_2 \end{cases}$$

A *well-weighted flow* is a global-flow specification such that for every sub-expression $\textbf{let } x = \mathcal{B}_0 \textbf{ in } \mathcal{B}$ in $\mathcal{A}$, and for every $x^n \in FV(\mathcal{B})$, we have $n > w(\mathcal{B}_0)$. Weights can be easily assigned to a flow so the resulting flow is well-weighted by assigning weights to flow variables from left to right (skipping weights where appropriate).

Lemma 2.4 (Reduction Decreases Weight). *Let $\mathcal{A}$ be a well-weighted global-flow specification, and $\mathcal{A}'$ a global-flow specification such that $\mathcal{A} \to \mathcal{A}'$, then $w(\mathcal{A}') < w(\mathcal{A})$.*

Proof. By enumerating possibilities of reduction. If $\mathcal{A} = \mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B}$ and $\mathcal{A}' = [x := \mathcal{B}_0]\mathcal{B}$, then it is the case that $w(\mathcal{A}') < w(\mathcal{A})$, since weights in $\mathcal{B}$ are erased and replaced by weights in $\mathcal{B}_0$. For all other cases, we use the induction hypothesis on the sub-expression being reduced, and it is the case that the sum of weights of sub-expressions decrease. □

Lemma 2.5 (Strong Normalization). *The notion of reduction "$\to$" is strongly normalizing, i.e., any global-flow specification $\mathcal{A}$ reduces to some $\mathcal{B}$ in finite number of steps.*

Proof. Since weight of a flow always decreases for every step of reduction, and that a weight cannot decrease below 0, it must be the case that reduction stops in a finite number of steps. □

Unique Normal Form

Lemma 2.6 (Confluence). *Let $\mathcal{A}$, $\mathcal{B}$, $\mathcal{B}'$ be global-flow specifications such that $\mathcal{A} \xrightarrow{*} \mathcal{B}$ and $\mathcal{A} \xrightarrow{*} \mathcal{B}'$, then we have $\mathcal{B} \xrightarrow{*} C$ and $\mathcal{B}' \xrightarrow{*} C$ for some global-flow specification C.*

Proof. This is implied by Weak Confluence (Lemma 2.3) and Strong Normalization (Lemma 2.5), a corollary of Newman's result [1]. □

Lemma 2.7 (Unique Normal Form). *Every global-flow specification $\mathcal{A}$ has a normal form, which is unique.*

Proof. This is implied by Confluence (Lemma 2.6) and Strong Normalization (Lemma 2.5). □

We presented in this section a language for global-flow specification, and showed that, although reduction of flows is non-deterministic, reduction strongly normalizes, and every flows have a unique normal form. Taking the view that reduction corresponds to the act of assembling flow controllers, this means assembly is a finite process (putting together flow controllers never takes forever), and the order of assembly never matters. This seemingly trivial and intuitive conclusion may not hold true for more-complex languages.

In the next section, we formalize the behavior in the notion of *types* and *typing rules*, both of which form the basis of type analysis.

3 Flow Types

In the specification language, types are invariants that describe the characteristics of flow components. The characteristics are abstracted over the level of wire, or *socket types*; over the level of bus, or plain *types*; and over the level of component, or *flow types*, which consist of input and output plain types in the forward and backward directions.

A *socket type* is a description of a single logical "entry" or "exit" point for a flow. The type itself is pertinent to the particular application, as those illustrated in [2].

In this paper, socket types are *first order*, meaning that these types are not parameterized over some variants. For example, the property of signal delay *cannot* be naively encoded by letting a flow's input type be quantified over some time-stamp t and its output type be some $t + \epsilon$ where ϵ is the delay caused internally by the flow component. Furthermore, socket types must have a *partial ordering*. More details are given in Section 3.2.

A plain *type* is a binary-tree construction of socket types or just a socket type alone, describing a composite socket that corresponds to entry and exit points to one or more parallel flows (as in Figure 2.1(a)).

A *flow type* is a 4-tuple representing both the forward and backward entry and exit points to a flow. This represents the complete type specification of a component in the flow specification. We will present these tuples graphically as two-by-two matrices so each element's position in the matrix corresponds with its graphical placement in the flow of Figure 2.1.

3.1 Syntax of Types

The syntax of types and the meta-variables ranging over their sorts are given by the following extended-BNF definition:

$$\begin{array}{lcll} r & \in & \mathsf{FwSocketType} & \\ s & \in & \mathsf{BwSocketType} & \\ t & \in & \mathsf{SocketType} & ::= \quad r \mid s \\ \rho & \in & \mathsf{FwType} & ::= \quad r \mid (\rho_1 \cdot \rho_2) \\ \sigma & \in & \mathsf{BwType} & ::= \quad s \mid (\sigma_1 \cdot \sigma_2) \\ \tau & \in & \mathsf{Type} & ::= \quad \rho \mid \sigma \\ T & \in & \mathsf{FlowType} & ::= \quad \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \end{array}$$

Note that $\mathsf{SocketType} \subset \mathsf{Type}$, but that $\mathsf{Type} \cap \mathsf{FlowType} = \varnothing$, i.e., it is safe to promote a socket type to a plain type but that no plain type (of itself) constitutes a flow type.

We reference the four components of a flow type with the operators f-in, f-out, b-out and b-in; namely, if

$$T = \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$$

then $\mathsf{f\text{-}in}(T) = \rho_1$, $\mathsf{f\text{-}out}(T) = \rho_2$, $\mathsf{b\text{-}out}(T) = \sigma_1$ and $\mathsf{b\text{-}in}(T) = \sigma_2$. Notice that the exact binary splitting of plain types in a connection from one flow's output to another flow's input need to match, but the flow itself may have asymmetric splitting on its input and output elements.

A useful operation is to take the product of two plain types (denoted "$\cdot$") and the product of two flow types (denoted "$\bullet$"). The product of two plain types appears in the syntax of FwType and BwType. The product of two flow types is defined below:

$$\begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \bullet \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} = \begin{bmatrix} (\rho_1 \cdot \rho_3) & (\rho_2 \cdot \rho_4) \\ (\sigma_1 \cdot \sigma_3) & (\sigma_2 \cdot \sigma_4) \end{bmatrix}$$

This flow type operation corresponds to parallel composition at the flow level.

3.2 Subtyping

In the previous section, we mentioned that socket types must have a *partial ordering*. Such ordering relation forms our basis of compatibility, i.e., an output socket having type t_1 is said to be compatible with an input socket type t_2 exactly when the property described by t_2 includes or supersedes that of t_1. In other words, t_1 is compatible with t_2 if and only if t_1 *relates* to t_2 by some subtyping assumption. We say that Δ is the set of such subtyping assumptions on forward and backward socket types,

$$\Delta \subseteq \mathsf{FwSocketType} \times \mathsf{FwSocketType} \cup \mathsf{BwSocketType} \times \mathsf{BwSocketType} \ ,$$

and that each individual relation is written as $r_1 <: r_2$ for some $r_1, r_2 \in \mathsf{FwSocketType}$ (read as "r_1 is a subtype of r_2") or as $s_1 <: s_2$ for some $s_1, s_2 \in \mathsf{BwSocketType}$ (read as "s_1 is a subtype of s_2"). Note that forward socket types do not relate to backward socket types and vice versa. Subtyping assumptions are given to us depending on the application—many examples of Δ can be found in [2]—but we require Δ to be a partial ordering, which gives us the following properties:

$$\frac{\{t_1 <: t_2\} \subseteq \Delta}{\Delta \vdash t_1 <: t_2}\ (\mathsf{stype}) \qquad \frac{t \in \mathsf{SocketType}}{\Delta \vdash t <: t}\ (\mathsf{stype\text{-}refl}) \qquad \frac{\Delta \vdash t_1 <: t_2 \quad \Delta \vdash t_2 <: t_3}{\Delta \vdash t_1 <: t_3}\ (\mathsf{stype\text{-}trans})$$

and antisymmetry, i.e., for all socket types t_1 and t_2, if $\Delta \vdash t_1 <: t_2$ and $\Delta \vdash t_2 <: t_1$, then $t_1 = t_2$. More specifically, antisymmetry defines our notion of Δ being *consistent*.

Assumption 3.1. *Δ is a fixed, but otherwise arbitrary, consistent set of subtyping assumptions throughout the rest of these notes.*

We extend the subtyping relation to types and flow types, using the following rules:

$$\frac{\Delta \vdash \rho_1 <: \rho_1' \quad \Delta \vdash \rho_2 <: \rho_2'}{\Delta \vdash (\rho_1 \cdot \rho_2) <: (\rho_1' \cdot \rho_2')}\ (\textsf{fwtype-lift}) \qquad \frac{\Delta \vdash \sigma_1 <: \sigma_1' \quad \Delta \vdash \sigma_2 <: \sigma_2'}{\Delta \vdash (\sigma_1 \cdot \sigma_2) <: (\sigma_1' \cdot \sigma_2')}\ (\textsf{bwtype-lift})$$

$$\frac{\Delta \vdash \rho_3 <: \rho_1 \quad \Delta \vdash \rho_2 <: \rho_4 \quad \Delta \vdash \sigma_1 <: \sigma_3 \quad \Delta \vdash \sigma_4 <: \sigma_2}{\Delta \vdash \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} <: \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}}\ (\textsf{ftype-lift})$$

Subtyping lifted to flow types (the last rule) is *contravariant* in the two types along the first diagonal and *covariant* in the two types along the second diagonal.

Using the lifting rules above, reflexivity and transitivity on types and flow types can be derived accordingly:

$$\frac{\tau \in \textsf{Type}}{\Delta \vdash \tau <: \tau}\ (\textsf{type-refl}) \qquad \frac{\Delta \vdash \tau_1 <: \tau_2 \quad \Delta \vdash \tau_2 <: \tau_3}{\Delta \vdash \tau_1 <: \tau_3}\ (\textsf{type-trans})$$

$$\frac{T \in \textsf{FlowType}}{\Delta \vdash T <: T}\ (\textsf{ftype-refl}) \qquad \frac{\Delta \vdash T_1 <: T_2 \quad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}\ (\textsf{ftype-trans})$$

These rules form the basis of our judgment. We can also show the antisymmetry of type and flow type.

Lemma 3.2 (Antisymmetry of Type**).** *Suppose the set Δ of subtyping assumptions is consistent. For all flow types τ_1 and τ_2, if both $\Delta \vdash \tau_1 <: \tau_2$ and $\Delta \vdash \tau_2 <: \tau_1$, then it must be that $\tau_1 = \tau_2$.*

Proof. First of all, the only way $\Delta \vdash \tau_1 <: \tau_2$ or $\Delta \vdash \tau_2 <: \tau_1$ is when either both $\tau_1 = t_1$ and $\tau_2 = t_2$ for some t_1, t_2, or both $\tau_1 = (\tau_{1,1} \cdot \tau_{1,2})$ and $\tau_2 = (\tau_{2,1} \cdot \tau_{2,2})$ for some $\tau_{1,1}, \tau_{1,2}, \tau_{2,1}, \tau_{2,2}$. In the former case, we reduce the problem on antisymmetry of socket type, which is an assumption. The latter case can be pursued by induction hypothesis on $\tau_{1,1}, \tau_{2,1}$ and $\tau_{1,2}, \tau_{2,2}$. □

Lemma 3.3 (Antisymmetry of FlowType**).** *Suppose the set Δ of subtyping assumptions is consistent. For all flow types T_1 and T_2, if both $\Delta \vdash T_1 <: T_2$ and $\Delta \vdash T_2 <: T_1$, then it must be that $T_1 = T_2$.*

Proof. Suppose $\Delta \vdash T_1 <: T_2$ and $\Delta \vdash T_2 <: T_1$, but $T_1 \neq T_2$. We proceed to prove by contradiction.

Let $T_1 = \begin{bmatrix} \rho_{1,1} & \rho_{1,2} \\ \sigma_{1,1} & \sigma_{1,2} \end{bmatrix}$, $T_2 = \begin{bmatrix} \rho_{2,1} & \rho_{2,2} \\ \sigma_{2,1} & \sigma_{2,2} \end{bmatrix}$, then, using the inference rule flow-lift, for $\Delta \vdash T_1 <: T_2$, we have $\Delta \vdash \rho_{2,1} <: \rho_{1,1}$, $\Delta \vdash \rho_{1,2} <: \rho_{2,2}$, $\Delta \vdash \sigma_{1,1} <: \sigma_{2,1}$, and $\Delta \vdash \sigma_{2,2} <: \sigma_{1,2}$; also for $\Delta \vdash T_2 <: T_1$, we have $\Delta \vdash \rho_{1,1} <: \rho_{2,1}$, $\Delta \vdash \rho_{2,2} <: \rho_{1,2}$, $\Delta \vdash \sigma_{2,1} <: \sigma_{1,1}$, and $\Delta \vdash \sigma_{1,2} <: \sigma_{2,2}$.

Since $T_1 \neq T_2$, at least one of the following must be the case:

$$\rho_{1,1} \neq \rho_{2,1},\ \rho_{1,2} \neq \rho_{2,2},\ \sigma_{1,1} \neq \sigma_{2,1},\ \sigma_{1,2} \neq \sigma_{2,2}$$

However, by Lemma 3.2,

- $\Delta \vdash \rho_{2,1} <: \rho_{1,1}$ and $\Delta \vdash \rho_{1,1} <: \rho_{2,1}$ contradicts with $\rho_{1,1} \neq \rho_{2,1}$.
- $\Delta \vdash \rho_{1,2} <: \rho_{2,2}$ and $\Delta \vdash \rho_{2,2} <: \rho_{1,2}$ contradicts with $\rho_{1,2} \neq \rho_{2,2}$.
- $\Delta \vdash \sigma_{1,1} <: \sigma_{2,1}$ and $\Delta \vdash \sigma_{2,1} <: \sigma_{1,1}$ contradicts with $\sigma_{1,1} \neq \sigma_{2,1}$.
- $\Delta \vdash \sigma_{2,2} <: \sigma_{1,2}$ and $\Delta \vdash \sigma_{1,2} <: \sigma_{2,2}$ contradicts with $\sigma_{1,2} \neq \sigma_{2,2}$.

So none of the inequality above holds, therefore it must be the case that $T_1 = T_2$. □

Since subtyping is used throughout the specification language, it is important that we can decide subtyping at all levels of abstraction efficiently. Deciding subtyping at a higher level is based on deciding subtyping at a lower level.

Lemma 3.4 (Deciding Subtyping Is Efficient). *If we can decide in constant time whether an arbitrary subtyping assumption $t_1 <: t_2$ holds, i.e., whether $\{t_1 <: t_2\} \subseteq \Delta$, then we can decide whether a subtyping judgment $\Delta \vdash \tau_1 <: \tau_2$ holds in time linear in the size of τ_1 and τ_2.*

Proof. Given by the inference rule on FwType and BwType, subtyping relation is decided by structurally decomposing the pair and compare the socket type one by one. If subtyping of socket type can be determined in constant time, then subtyping of FwType and BwType can also be determined in linear time, since every socket type is compared exactly once. □

Figure 4.1: Typing Rules for Flows

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\mathsf{var}) \qquad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \quad \Delta \vdash T <: T'}{\Gamma, \Delta \vdash \mathcal{A} : T'}\ (\mathsf{sub})$$

$$\frac{\mathsf{type}(A) = T}{\Gamma, \Delta \vdash A : T}\ (\mathsf{local}) \qquad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \quad \Gamma, \Delta \vdash \mathcal{B} : T'}{\Gamma, \Delta \vdash \mathcal{A} \| \mathcal{B} : T \bullet T'}\ (\mathsf{par})$$

$$\frac{\Gamma, \Delta \vdash \mathcal{A} : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash \mathcal{B} : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 <: \rho_3 \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{A}; \mathcal{B} : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}}\ (\mathsf{seq})$$

$$\frac{\Gamma, \Delta \vdash \mathcal{A} : T \quad \Gamma \cup \{x : T\}, \Delta \vdash \mathcal{B} : T'}{\Gamma, \Delta \vdash \mathbf{let}\ x = \mathcal{A}\ \mathbf{in}\ \mathcal{B} : T'}\ (\mathsf{let})$$

4 Typing Rules

Assigning types to flows and their constituent elements enables us to abstract away the internals of each component and perform analysis to assess whether the pieces of a global flow specification interact according to the declared composition structures, i.e., whether the pieces "fit" together, and what "shape" any gaps (flow variables) in the specification might have. This is done in two stages: first, we specify typing rules which formally encode how types can be assigned to global flows as they are built up from local flows, flow variables, and other global flows; second, we discuss the formulation of practical algorithms for deducing these conclusions from a specification provided by the system architect or programmer.

Figure 4.1 presents a formal declaration of rules for typing global flows based upon complete or partial knowledge of the types of their constituent flows.

The typing rules presented in Figure 4.1 are used to derive *judgments* of the form $\Gamma, \Delta \vdash \mathcal{A} : T$ where Γ is a type environment, Δ is the set of subtyping assumptions discussed in Section 3.2, $\mathcal{A}$ is a global-flow specification, and T is a flow type. We say that $\mathcal{A}$ *type checks* if there is a type environment Γ and a flow type T such that the rules in Figure 4.1 can be applied to derive the conclusion that, if we assume Γ, $\mathcal{A}$ has the flow type T.

A type environment Γ is a partial function from the set $\mathsf{FlowVar}$ of flow variables to the set $\mathsf{FlowType}$ of flow types. If Γ has a finite domain of definition, we write it as

$$\Gamma = \{x_1 : T_1,\ \ldots\ ,\ x_n : T_n\}$$

where $n \geqslant 0$, for some pairwise distinct flow variables $x_1,\ \ldots,\ x_n$ and some flow types $T_1,\ \ldots,\ T_n$.

We assume the existence of a total function type from the set of all local flows to the set of flow types, $\mathsf{type} : \mathsf{LocalFlow} \to \mathsf{FlowType}$.

Relative to Δ (recall that Δ is a set of subtyping assumptions), we say some global-flow specification $\mathcal{A}$ *type checks* if there is a type environment Γ and a flow type T such that the judgment $\Gamma, \Delta \vdash \mathcal{A} : T$ holds, i.e., it is derivable using the aforementioned typing rules.

It is also important that, if $\mathcal{A}$ can still be reduced further, then the resulting specification preserves the same behavior, i.e., the flow type is preserved. This is formally known as Type Preservation theorem. For this, we first prove the Substitution lemma, which shows that flow type is preserved when a variable is substituted for a specification with compatible flow type. We then use Substitution lemma in the proof of type preservation, in particular the **let** case, where substitution would occur.

Lemma 4.1 (Substitution). *If* $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : T$ *and* $\Gamma, \Delta \vdash \mathcal{A} : T_0$, *then* $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{B} : T$.

Proof. By analyzing the possible forms of global-flow specification $\mathcal{B}$.

- case $\mathcal{B}$ is y, with $y : T \in \Gamma \cup \{x : T_0\}$. If $x = y$, then $T = T_0$, and that $[x := \mathcal{A}]y = \mathcal{A}$, so $\Gamma, \Delta \vdash [x := \mathcal{A}]y : T$ is the same as $\Gamma, \Delta \vdash \mathcal{A} : T_0$, which is given. Otherwise, if $x \neq y$, then $[x := \mathcal{A}]y = y$, and that $\Gamma, \Delta \vdash y : T$ is immediate.

- case $\mathcal{B}$ is A. $[x := \mathcal{A}]A = A$, so the result is immediate.
- case $\mathcal{B}$ is $\mathcal{C}_1; \mathcal{C}_2$ where, by the premises of the typing rule, we have $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ and $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$, and that $\Delta \vdash \rho_2 <: \rho_3$ and $\Delta \vdash \sigma_3 <: \sigma_2$ so $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$ is derivable. Suppose $\Gamma, \Delta \vdash \mathcal{A} : T_0$, then by induction hypothesis on the premises, we have $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ and $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$, so $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{B} : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$ is derivable.
- case $\mathcal{B}$ is $\mathcal{C}_1 \| \mathcal{C}_2$ where $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_1 : T_1$ and $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_2 : T_2$ are the premises of the typing rule, so that $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : T_1 \bullet T_2$ is derivable. Suppose $\Gamma, \Delta \vdash \mathcal{A} : T_0$, then by induction hypothesis on the premises, we have $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_1 : T_1$ and $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_2 : T_2$ respectively, so $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{B} : T_1 \bullet T_2$ is derivable.
- case $\mathcal{B}$ is $\mathbf{let}\ \ y = \mathcal{C}_0\ \ \mathbf{in}\ \ \mathcal{C}$.

  First, when $x \neq y$, we have $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_0 : T'$ and $\Gamma \cup \{x : T_0, y : T'\}, \Delta \vdash \mathcal{C} : T$ from the premises of typing rule so that $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathbf{let}\ \ y = \mathcal{C}_0\ \ \mathbf{in}\ \ \mathcal{C} : T$ is derivable. Suppose $\Gamma, \Delta \vdash \mathcal{A} : T_0$, then by induction hypothesis on the premises, we have both $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_0 : T'$ and $\Gamma \cup \{y : T'\}, \Delta \vdash [x := \mathcal{A}]\mathcal{C} : T$, so $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathbf{let}\ \ y = \mathcal{C}_0\ \ \mathbf{in}\ \ \mathcal{C} : T$ is derivable.

  When $x = y$, then we have $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{C}_0 : T'$ and $\Gamma \cup \{x : T'\}, \Delta \vdash \mathcal{C} : T$ as the premises. Suppose $\Gamma, \Delta \vdash \mathcal{A} : T_0$, then by induction hypothesis on the first premise, we have $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{C}_0 : T'$. Then $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathbf{let}\ \ y = \mathcal{C}_0\ \ \mathbf{in}\ \ \mathcal{C} : T$ is the same as $\Gamma, \Delta \vdash \mathbf{let}\ \ y = [x := \mathcal{A}]\mathcal{C}_0\ \ \mathbf{in}\ \ \mathcal{C} : T$, which is derivable.

□

Theorem 4.2 (Soundness). *If the judgment $\Gamma, \Delta \vdash \mathcal{A} : T$ is derivable and $\mathcal{A} \to \mathcal{A}'$, then $\Gamma, \Delta \vdash \mathcal{A}' : T$ is also derivable. In particular, if $\mathcal{A}$ type checks, then so does $\mathcal{A}'$, and both $\mathcal{A}$ and $\mathcal{A}'$ have the same type.*

Proof. By analyzing the possible forms of global-flow specification $\mathcal{A}$ and reduction rules from $\mathcal{A}$ to $\mathcal{A}'$.

- case $\mathcal{A}$ is x, which cannot be reduced further, so soundness immediately follows for this case.
- case $\mathcal{A}$ is A, which also cannot be reduced further, so soundness immediately follows for this case.
- case $\mathcal{A}$ is $\mathcal{B}_1; \mathcal{B}_2$, then by the premises of the typing rule (seq) for $\Gamma, \Delta \vdash \mathcal{B}_1; \mathcal{B}_2 : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$ to be derivable, we have $\Gamma, \Delta \vdash \mathcal{B}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ and $\Gamma, \Delta \vdash \mathcal{B}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$, and that $\Delta \vdash \rho_2 <: \rho_3$ and $\Delta \vdash \sigma_3 <: \sigma_2$. According to reduction rules, we have two possibilities.
  - sub-case $\mathcal{A}'$ is $\mathcal{B}_1'; \mathcal{B}_2$ where $\mathcal{B}_1 \to \mathcal{B}_1'$. By induction hypothesis on $\mathcal{B}_1$, we have $\Gamma, \Delta \vdash \mathcal{B}_1' : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$, so we construct a derivation for $\Gamma, \Delta \vdash \mathcal{B}_1'; \mathcal{B}_2 : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_1' : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash \mathcal{B}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 <: \rho_3 \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{B}_1'; \mathcal{B}_2 : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}}$$

– sub-case $\mathcal{A}'$ is $\mathcal{B}_1; \mathcal{B}_2'$ where $\mathcal{B}_2 \rightarrow \mathcal{B}_2'$. By induction hypothesis on $\mathcal{B}_2$, we have and $\Gamma, \Delta \vdash \mathcal{B}_2' : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$, so we construct a derivation for $\Gamma, \Delta \vdash \mathcal{B}_1; \mathcal{B}_2' : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash \mathcal{B}_2' : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 <: \rho_3 \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{B}_1; \mathcal{B}_2' : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}}$$

- case $\mathcal{A}$ is $\mathcal{B}_1 \| \mathcal{B}_2$, then by the premises of the typing rule (par) where $\Gamma, \Delta \vdash \mathcal{B}_1 \| \mathcal{B}_2 : T_1 \bullet T_2$ is derivable, we have $\Gamma, \Delta \vdash \mathcal{B}_1 : T_1$ and $\Gamma, \Delta \vdash \mathcal{B}_2 : T_2$. According to reduction rules, we have two possibilities.

  – sub-case $\mathcal{A}'$ is $\mathcal{B}_1' \| \mathcal{B}_2$ where $\mathcal{B}_1 \rightarrow \mathcal{B}_1'$. By induction hypothesis on $\mathcal{B}_1$, we have $\Gamma, \Delta \vdash \mathcal{B}_1' : T_1$, so we construct a derivation for $\Gamma, \Delta \vdash \mathcal{B}_1' \| \mathcal{B}_2 : T_1 \bullet T_2$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_1' : T_1 \quad \Gamma, \Delta \vdash \mathcal{B}_2 : T_2}{\Gamma, \Delta \vdash \mathcal{B}_1' \| \mathcal{B}_2 : T_1 \bullet T_2}$$

  – sub-case $\mathcal{A}'$ is $\mathcal{B}_1 \| \mathcal{B}_2'$ where $\mathcal{B}_2 \rightarrow \mathcal{B}_2'$. By induction hypothesis on $\mathcal{B}_2$, we have $\Gamma, \Delta \vdash \mathcal{B}_2 : T_2$, so we construct a derivation for $\Gamma, \Delta \vdash \mathcal{B}_1 \| \mathcal{B}_2' : T_1 \bullet T_2$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_1 : T_1 \quad \Gamma, \Delta \vdash \mathcal{B}_2' : T_2}{\Gamma, \Delta \vdash \mathcal{B}_1 \| \mathcal{B}_2' : T_1 \bullet T_2}$$

- case $\mathcal{A}$ is $\mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B}$, suppose $\Gamma, \Delta \vdash \mathcal{B}_0 : T_0$ and that $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : T$ by the premises of typing rule (let) so that $\Gamma, \Delta \vdash \mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B} : T$ is derivable. Using Lemma 4.1, $\Gamma, \Delta \vdash [x := \mathcal{B}_0]\mathcal{B} : T$ is derivable.

□

A lesser-known notion of type completeness can also be proved for the language of global-flow specifications.

Lemma 4.3 (Inverse Substitution Lemma). *If* $\Gamma, \Delta \vdash [x := \mathcal{A}]\mathcal{B} : T$ *and* $\Gamma, \Delta \vdash \mathcal{A} : T_0$*, then* $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : T$.

Theorem 4.4 (Completeness). *If the judgment* $\Gamma, \Delta \vdash \mathcal{A}' : T$ *is derivable, and* $\mathcal{A} \rightarrow \mathcal{A}'$ *for some global-flow specification* $\mathcal{A}$*, then* $\Gamma, \Delta \vdash \mathcal{A} : T$ *is also derivable. That is, if* $\mathcal{A}'$ *type checks, then so does all possible* $\mathcal{A}$ *that can be reduced to* $\mathcal{A}'$*, and that both* $\mathcal{A}$ *and* $\mathcal{A}'$ *have the same type.*

Proof. By induction on cases of $\mathcal{A}'$. For simplicity, we assume that $\mathcal{A} \neq \mathcal{A}'$. For the possibilities of $\mathcal{A}$ that reduces to $\mathcal{A}'$, we refer to the reduction rules back in Section 2.2.

- case $\mathcal{A}'$ is $\mathcal{A}_1'; \mathcal{A}_2'$, and suppose both $\Gamma, \Delta \vdash \mathcal{A}_1' : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ and $\Gamma, \Delta \vdash \mathcal{A}_2' : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$ are derivable, such that $\Delta \vdash \rho_2 <: \rho_3$ and $\Delta \vdash \sigma_3 <: \sigma_2$. By typing rule (seq), $T = \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}$. We have several possibilities of $\mathcal{A}$ according to reduction rules.

  – sub-case $\mathcal{A}$ is $\mathcal{A}_1; \mathcal{A}_2'$, where $\mathcal{A}_1 \rightarrow \mathcal{A}_1'$. By induction hypothesis, $\Gamma, \Delta \vdash \mathcal{A}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathcal{A}_1; \mathcal{A}_2' : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{A}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash \mathcal{A}_2' : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 <: \rho_3 \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{A}_1; \mathcal{A}_2' : T}$$

– sub-case $\mathcal{A}$ is $\mathcal{A}_1'; \mathcal{A}_2$, where $\mathcal{A}_2 \to \mathcal{A}_2'$. By induction hypothesis, $\Gamma, \Delta \vdash \mathcal{A}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathcal{A}_1'; \mathcal{A}_2 : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{A}_1' : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash \mathcal{A}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 <: \rho_3 \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{A}_1'; \mathcal{A}_2 : T}$$

- case $\mathcal{A}'$ is $\mathcal{A}_1' \| \mathcal{A}_2'$, and suppose both $\Gamma, \Delta \vdash \mathcal{A}_1' : T_1$ and $\Gamma, \Delta \vdash \mathcal{A}_2' : T_2$ are derivable. By typing rule (par), $T = T_1 \bullet T_2$. We have several possibilities of $\mathcal{A}$ according to reduction rules.

  – sub-case $\mathcal{A}$ is $\mathcal{A}_1 \| \mathcal{A}_2'$, where $\mathcal{A}_1 \to \mathcal{A}_1'$. By induction hypothesis, $\Gamma, \Delta \vdash \mathcal{A}_1 : T_1$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathcal{A}_1 \| \mathcal{A}_2' : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{A}_1 : T_1 \quad \Gamma, \Delta \vdash \mathcal{A}_2' : T_2}{\Gamma, \Delta \vdash \mathcal{A}_1 \| \mathcal{A}_2' : T}$$

  – sub-case $\mathcal{A}$ is $\mathcal{A}_1' \| \mathcal{A}_2$, where $\mathcal{A}_2 \to \mathcal{A}_2'$. By induction hypothesis, $\Gamma, \Delta \vdash \mathcal{A}_2 : T_2$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathcal{A}_1' \| \mathcal{A}_2 : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{A}_1' : T_1 \quad \Gamma, \Delta \vdash \mathcal{A}_2 : T_2}{\Gamma, \Delta \vdash \mathcal{A}_1' \| \mathcal{A}_2 : T}$$

- case $\mathcal{A}'$ is $\mathbf{let}\ \ x = \mathcal{B}_0'\ \ \mathbf{in}\ \ \mathcal{B}'$, and suppose both $\Gamma, \Delta \vdash \mathcal{B}_0' : T_0$ and $\Gamma \cup \{x : T_0\} \vdash \mathcal{B}' : T$ are derivable. We have several possibilities of $\mathcal{A}$ according to reduction rules.

  – sub-case $\mathcal{A}$ is $\mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B}'$, where $\mathcal{B}_0 \to \mathcal{B}_0'$. By induction hypothesis, $\Gamma, \Delta \vdash \mathcal{B}_0 : T_0$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B}' : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_0 : T_0 \quad \Gamma \cup \{x : T_0\} \vdash \mathcal{B}' : T}{\Gamma, \Delta \vdash \mathbf{let}\ \ x = \mathcal{B}_0\ \ \mathbf{in}\ \ \mathcal{B}' : T}$$

  – sub-case $\mathcal{A}$ is $\mathbf{let}\ \ x = \mathcal{B}_0'\ \ \mathbf{in}\ \ \mathcal{B}$, where $\mathcal{B} \to \mathcal{B}'$. By induction hypothesis, $\Gamma \cup \{x : T_0\}, \Delta \vdash \mathcal{B} : T$ is derivable. We construct a derivation for $\Gamma, \Delta \vdash \mathbf{let}\ \ x = \mathcal{B}_0'\ \ \mathbf{in}\ \ \mathcal{B} : T$ as follows,

$$\frac{\Gamma, \Delta \vdash \mathcal{B}_0' : T_0 \quad \Gamma \cup \{x : T_0\} \vdash \mathcal{B} : T}{\Gamma, \Delta \vdash \mathbf{let}\ \ x = \mathcal{B}_0'\ \ \mathbf{in}\ \ \mathcal{B} : T}$$

- For other cases where $\mathcal{A} \to \mathcal{A}'$ is reduced by substitution, we use the Inverse Substitution Lemma (Lemma 4.3).

□

Now with both type soundness and completeness, we revisit the claim that was made in Section 2.2, that if $\mathcal{A}$ and $\mathcal{B}$ are both typable and are in the same equivalence class, then $\mathcal{A}$ and $\mathcal{B}$ would have the same type.

4.1 Equivalence of Typability and Safety

We first formalize the notion of global-flow equivalence classes.

Definition 4.5 (Equivalence Relation). Let $\sim$ be the equivalence relation generated by $\xrightarrow{*}$, our notion of reduction in reflexive, transitive closure:

1. $\mathcal{A} \xrightarrow{*} \mathcal{B}$ implies $\mathcal{A} \sim \mathcal{B}$,
2. $\mathcal{A} \sim \mathcal{B}$ implies $\mathcal{B} \sim \mathcal{A}$,
3. $\mathcal{A} \sim \mathcal{C}$ and $\mathcal{C} \sim \mathcal{B}$ imply $\mathcal{A} \sim \mathcal{B}$.

Lemma 4.6. *Let $\mathcal{A}$, $\mathcal{B}$ be global-flow specifications. If $\Gamma, \Delta \vdash \mathcal{A} : T$ is derivable and $\mathcal{A} \sim \mathcal{B}$, then $\Gamma, \Delta \vdash \mathcal{B} : T$ is also derivable.*

Proof. This immediately follows from type soundness (Theorem 4.2) and type completeness (Theorem 4.4). □

Corollary 4.7 (Typability and Safety are Equivalent). *A global flow specification $\mathcal{A}$ is typable if and only if its normal form is typable.*

Proof. Suppose $\mathcal{A}$ normalizes to $\mathcal{B}$. Clearly $\mathcal{A} \sim \mathcal{B}$, and by Lemma 4.6, if $\mathcal{A}$ is typable, then $\mathcal{B}$ is typable. By definition of $\sim$, it is also the case that $\mathcal{B} \sim \mathcal{A}$, so it follows that if $\mathcal{B}$ is typable, then so as $\mathcal{A}$. □

What it means by a normal form to be typable is that, for all flow connections down to the socket level, a signal that comes from some output socket at least satisfies the requirement of an input socket. Corollary 4.7 shows that the typing rules in Figure 4.1 capture exactly this property.

4.2 Syntax-directed Type Checking

At this point, we have a framework that consists of a syntax for global-flow specification, a syntax for types, assumptions on subtyping relation, and a set of type judgment rules for the global-flow specification. Within this framework, a judgment can be decided about any closed specification. This gives rise to an analysis algorithm that checks a global-flow specification against the framework, the *Non-compositional Analysis*, or $\mathcal{P}_{NC}$.

The algorithm kicks start the subroutine $\mathcal{P}'_{NC}$, as shown in Figure 4.2, to syntactically decompose a specification recursively and apply appropriate type judgment rules, with the initial condition that the type environment Γ is empty. If the specification can be typed, the final judgment is returned. Otherwise the algorithm fails with an error condition.

The systematic chase of type judgment makes it straightforward to see (and prove) that non-compositional analysis algorithm adheres to the type judgment rules.

Figure 4.2: Non-compositional Analysis Algorithm $\mathcal{P}_{NC}$

$\mathcal{P}_{NC}(\mathcal{A}) = \mathcal{P}'_{NC}(\varnothing, \mathcal{A})$
$\mathcal{P}'_{NC}(\Gamma, \mathcal{A}) =$

1. **IF** $\mathcal{A} = A$ **THEN RETURN** $\mathsf{type}(\mathcal{A})$.
2. **IF** $\mathcal{A} = x$ **THEN RETURN** $\Gamma(x)$.
3. **IF** $\mathcal{A} = (\mathcal{A}_1; \mathcal{A}_2)$ **THEN**
   LET $T_1 = \mathcal{P}'_{NC}(\Gamma, \mathcal{A}_1)$,
   $T_2 = \mathcal{P}'_{NC}(\Gamma, \mathcal{A}_2)$
   IN **IF** $\Delta \vdash \mathsf{f\text{-}out}(T_1) <: \mathsf{f\text{-}in}(T_2)$ **AND**
   $\Delta \vdash \mathsf{b\text{-}out}(T_2) <: \mathsf{b\text{-}in}(T_1)$
   THEN **LET** $T = \begin{bmatrix} \mathsf{f\text{-}in}(T_1) & \mathsf{f\text{-}out}(T_2) \\ \mathsf{b\text{-}out}(T_1) & \mathsf{b\text{-}in}(T_2) \end{bmatrix}$
   IN **RETURN** T
   ELSE **FAILWITH** 'no solution' .
4. **IF** $\mathcal{A} = (\mathcal{A}_1 \| \mathcal{A}_2)$ **THEN**
   LET $T_1 = \mathcal{P}'_{NC}(\Gamma, \mathcal{A}_1)$,
   $T_2 = \mathcal{P}'_{NC}(\Gamma, \mathcal{A}_2)$
   IN **LET** $T = T_1 \bullet T_2$
   IN **RETURN** T .
5. **IF** $\mathcal{A} = (\mathbf{let}\ x = \mathcal{B}_0\ \mathbf{in}\ \mathcal{B})$ **THEN**
   LET $T_0 = \mathcal{P}'_{NC}(\Gamma, \mathcal{B}_0)$
   IN **LET** $T = \mathcal{P}'_{NC}(\Gamma \cup \{x : T_0\}, \mathcal{B})$
   IN **RETURN** T .

5 Conclusion

We presented a formalism for safe composition of networked controllers by introducing a language, a type system, and a type inference algorithm. The language is shown to be *strongly normalizing*, and that every specifications of global-flow have a unique normal form. The type system is shown to be equivalent to checking safety of compositions made only out of local flows. A short algorithm is also given to implement the typing rules.

The equivalence of typability and safety allows a flow programmer to use let-binding constructs for abstraction of flows, knowing that the result is the same as if she uses the composition in many places of a larger flow. Furthermore, let-binding provides a ground work to isolate the design aspect and usage aspect of flows—flows designed by one team can be "used" by another flow devised by the other team, and they are seamlessly integrated by the notion of flow variable substitution.

A shortcoming of the syntax-directed type checking algorithm is that we require the specification to contain no free flow variables. This means that, prior to the analysis, all flows to be filled to some gaps need to be given. This requirement is sometimes inconvenient, for example, during prototyping, when a team wants to find out if there are internal inconsistencies even if some flows are left open. Another example is when we only want to know whether some other flow will fit into the gaps, but revealing the actual flow composition is undesirable. This shortcoming is being addressed in a related work, the compositional analysis.

References

[1] H. P. (Hendrik Pieter) Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science B.V., paperback edition, 1985.

[2] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Safe Compositional Specification of Networking Systems. Technical Report BUCS-TR-2004-021, CS Department, Boston University, May 14 2004.

[3] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Typed Abstraction of Complex Network Compositions. Technical Report BUCS-TR-2005-014, CS Department, Boston University, May 1 2005.